Documentation

The Java™ Tutorials

Trail: Learning the Java Language
Lesson: Classes and Objects
Section: Nested Classes

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

Anonymous Classes

Anonymous classes enable you to make your code more concise. They enable you to declare and instantiate a class at the same time. They are like local classes except that they do not have a name. Use them if you need to use a local class only once.

This section covers the following topics:

- Declaring Anonymous Classes
- Syntax of Anonymous Classes
- Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class
- Examples of Anonymous Classes

Declaring Anonymous Classes

While local classes are class declarations, anonymous classes are expressions, which means that you define the class in another expression. The following example, `HelloWorldAnonymousClasses`, uses anonymous classes in the initialization statements of the local variables `frenchGreeting` and `spanishGreeting`, but uses a local class for the initialization of the variable `englishGreeting`:

```
public class HelloWorldAnonymousClasses {

    interface HelloWorld {
        public void greet();
        public void greetSomeone(String someone);
    }

    public void sayHello() {

        class EnglishGreeting implements HelloWorld {
            String name = "world";
            public void greet() {
                greetSomeone("world");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hello " + name);
            }
        }

        HelloWorld englishGreeting = new EnglishGreeting();

        HelloWorld frenchGreeting = new HelloWorld() {
            String name = "tout le monde";
            public void greet() {
                greetSomeone("tout le monde");
            }
            public void greetSomeone(String someone) {
                name = someone;
```

```
                System.out.println("Salut " + name);
            }
        };

        HelloWorld spanishGreeting = new HelloWorld() {
            String name = "mundo";
            public void greet() {
                greetSomeone("mundo");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hola, " + name);
            }
        };
        englishGreeting.greet();
        frenchGreeting.greetSomeone("Fred");
        spanishGreeting.greet();
    }

    public static void main(String... args) {
        HelloWorldAnonymousClasses myApp =
            new HelloWorldAnonymousClasses();
        myApp.sayHello();
    }
}
```

Syntax of Anonymous Classes

As mentioned previously, an anonymous class is an expression. The syntax of an anonymous class expression is like the invocation of a constructor, except that there is a class definition contained in a block of code.

Consider the instantiation of the `frenchGreeting` object:

```
HelloWorld frenchGreeting = new HelloWorld() {
    String name = "tout le monde";
    public void greet() {
        greetSomeone("tout le monde");
    }
    public void greetSomeone(String someone) {
        name = someone;
        System.out.println("Salut " + name);
    }
};
```

The anonymous class expression consists of the following:

- The new operator
- The name of an interface to implement or a class to extend. In this example, the anonymous class is implementing the interface `HelloWorld`.
- Parentheses that contain the arguments to a constructor, just like a normal class instance creation expression. **Note**: When you implement an interface, there is no constructor, so you use an empty pair of parentheses, as in this example.
- A body, which is a class declaration body. More specifically, in the body, method declarations are allowed but statements are not.

Because an anonymous class definition is an expression, it must be part of a statement. In this example, the anonymous class expression is part of the statement that instantiates the `frenchGreeting` object. (This explains why there is a semicolon after the closing brace.)

Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class

Like local classes, anonymous classes can capture variables; they have the same access to local variables of the enclosing scope:

- An anonymous class has access to the members of its enclosing class.
- An anonymous class cannot access local variables in its enclosing scope that are not declared as `final` or effectively final.
- Like a nested class, a declaration of a type (such as a variable) in an anonymous class shadows any other declarations in the enclosing scope that have the same name. See Shadowing for more information.

Anonymous classes also have the same restrictions as local classes with respect to their members:

- You cannot declare static initializers or member interfaces in an anonymous class.
- An anonymous class can have static members provided that they are constant variables.

Note that you can declare the following in anonymous classes:

- Fields
- Extra methods (even if they do not implement any methods of the supertype)
- Instance initializers
- Local classes

However, you cannot declare constructors in an anonymous class.

Examples of Anonymous Classes

Anonymous classes are often used in graphical user interface (GUI) applications.

Consider the JavaFX example `HelloWorld.java` (from the section Hello World, JavaFX Style from Getting Started with JavaFX). This sample creates a frame that contains a **Say 'Hello World'** button. The anonymous class expression is highlighted:

```
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World!");
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```

In this example, the method invocation `btn.setOnAction` specifies what happens when you select the **Say 'Hello World'** button. This method requires an object of type `EventHandler<ActionEvent>`. The `EventHandler<ActionEvent>` interface contains only one method, handle. Instead of implementing this method with a new class, the example uses an anonymous class expression. Notice that this expression is the argument passed to the `btn.setOnAction` method.

Because the `EventHandler<ActionEvent>` interface contains only one method, you can use a lambda expression instead of an anonymous class expression. See the section Lambda Expressions for more information.

Anonymous classes are ideal for implementing an interface that contains two or more methods. The following JavaFX example is from the section Customization of UI Controls. The highlighted code creates a text field that only accepts numeric values. It redefines the default implementation of the `TextField` class with an anonymous class by overriding the `replaceText` and `replaceSelection` methods inherited from the `TextInputControl` class.

```
    import javafx.application.Application;
    import javafx.event.ActionEvent;
    import javafx.event.EventHandler;
    import javafx.geometry.Insets;
```

```
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class CustomTextFieldSample extends Application {

    final static Label label = new Label();

    @Override
    public void start(Stage stage) {
        Group root = new Group();
        Scene scene = new Scene(root, 300, 150);
        stage.setScene(scene);
        stage.setTitle("Text Field Sample");

        GridPane grid = new GridPane();
        grid.setPadding(new Insets(10, 10, 10, 10));
        grid.setVgap(5);
        grid.setHgap(5);

        scene.setRoot(grid);
        final Label dollar = new Label("$");
        GridPane.setConstraints(dollar, 0, 0);
        grid.getChildren().add(dollar);

        final TextField sum = new TextField() {
            @Override
            public void replaceText(int start, int end, String text) {
                if (!text.matches("[a-z, A-Z]")) {
                    super.replaceText(start, end, text);
                }
                label.setText("Enter a numeric value");
            }

            @Override
            public void replaceSelection(String text) {
                if (!text.matches("[a-z, A-Z]")) {
                    super.replaceSelection(text);
                }
            }
        };

        sum.setPromptText("Enter the total");
        sum.setPrefColumnCount(10);
        GridPane.setConstraints(sum, 1, 0);
        grid.getChildren().add(sum);

        Button submit = new Button("Submit");
        GridPane.setConstraints(submit, 2, 0);
        grid.getChildren().add(submit);

        submit.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent e) {
                label.setText(null);
            }
        });

        GridPane.setConstraints(label, 0, 1);
        GridPane.setColumnSpan(label, 3);
        grid.getChildren().add(label);

        scene.setRoot(grid);
```